IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

United States Patent Application for

METHODS AND APPARATUS FOR GRAMMAR-BASED RECOGNITION OF USER-INTERFACE OBJECTS IN HTML APPLICATIONS

Inventors:

Name:	Attila Szepesvary
Address:	H-1151 Budapest, Szlacsanyi u. 186, Hungary
Citizenship:	Hungarian
Name:	Gergely Szilvasy
Address:	600 American Avenue, apt. C101, King of Prussia, PA 19406, USA
Citizenship:	Hungarian
Name:	Sandor Banki-Horvath
Address:	H-1118 Budapest, Torbagy u. 14. IV/14, Hungary
Citizenship:	Hungarian
Name:	Tamas Szvitacs
Address:	H-2094 Nagykovacsi, Banya u. 9, Hungary
Citizenship:	Hungarian

METHODS AND APPARATUS FOR GRAMMAR-BASED RECOGNITION OF USER-INTERFACE OBJECTS IN HTML APPLICATIONS

Reference to Related Application

This application claims the benefit of priority of U.S.S.N. 60/258,081, filed on December 22, 2000, entitled "METHODS AND APPARATUS FOR GRAMMAR-BASED RECOGNITION OF USER-INTERFACE OBJECTS IN HTML APPLICATIONS," the teachings of which are incorporated herein by reference.

Background of the Invention

The invention pertains to computer graphical user interface software and, particularly, to the automated recognition of objects presented by such interfaces. The invention has application (by way of non-limiting example) in training aids, accessibility aids, and quality assurance for stand-alone software applications, as well as for Internet- and other computer network-based software applications.

Though the computer was invented over forty years ago, use of the ubiquitous device remains a challenge for most people. Bound paper manuals gave way to on-line help as the computer went mainstream in the early 1980's. Software publishers have since struggled to make context-sensitive help screens that are comprehensive yet simple, thereby meeting the demands of novices and seasoned users alike.

On another front, software publishers must continually check their releases to assure that program displays are complete and consistent, particularly, where user input or response is required. Only through such quality assurance can the publishers be confident that their software will be understood by users.

Responding to the needs of sight-impaired users presents a host of additional problems. Basic program screens and help displays may need to be magnified, visually revamped or played in audio. The size of the quality assurance task may, consequently, double.

These problems are further compounded by the decreased use of stand-alone applications and the corresponding increased reliance on remote applications that "execute" via users' web browsers. Solutions to many of the aforementioned problems applicable to local applications often prove inapplicable to remotely executed ones.

An object of this invention is to provide improved methods and apparatus for use of graphical user interfaces.

A more particular object is to provide such methods and apparatus as can be used with software training aids, accessibility aids, and quality assurance for Internet- and other computer network-based software applications, as well as for stand-alone local software applications.

A still further object is to provide such methods and apparatus as can be used with software applications that utilize HTML-based and other markup language-based graphical user interfaces.

A related object is to provide such methods as can be readily implemented at low-cost, and without consumption of undue processor or memory resources and with adequate performance.

Summary of the Invention

The foregoing are among the objects attained by the invention, which provides methods and apparatus for grammar-based recognition of graphical user interface (UI) objects in an application executing on a web browser. This is achieved, according to one aspect of the invention, by parsing the application's underlying HTML (or other markup language) stream or document object model (DOM) in view of an application-specific grammar. Methods according to the invention can identify UI objects such as user input or selection fields, hyperlinks, and so forth, as well as properties of those objects, such as identifiers, content, shapes, locations, etc. Except where otherwise necessitated by context, the term UI object refers to objects, their identifiers and/or descriptors.

In a related aspect, the invention provides methods as described above that include scanning the HTML stream or DOM to identify tokens. These are parsed, based on the predefined grammar, to distinguish among UI objects as distinct from textual titles, metatags, unprintable HTML, in-line pictures and other "non-objects."

By way of non-limiting example, a system operating in accord with this aspect of the invention can be used to recognize customer name, home address, and other input fields (as well their content) displayed by a web browser (such as Microsoft Internet Explorer) while "pointing at" an Internet auction site web page. To this end, the system accesses the DOM created by the browser from the HTML stream representing that page. It identifies the aforementioned input fields by interpreting tokens discerned from the DOM in view of a grammar previously defined for that given application. Information regarding the position and content of those fields can be output, e.g., to a context-based "help" utility, or otherwise. Thus, by way of non-limiting example, that information can be used in a system of the type modeled after the OnDemand Personal Navigator training system, commercially available from the assignee hereof.

Further aspects of the invention provide methods as described above that include scanning the DOM hierarchy maintained in a browser in response to an HTML stream, and creating a sequence of tokens for each DOM element. This can include grouping sequences of

HTML DOM attributes into units (tokens) to create a mapping between the DOM elements and the tokens.

Further aspects of the invention provide methods as described above in which the tokens are parsed to find the HTML document structure as described by its predefined grammar. This can include grouping the tokens into syntactic structures that identify items displayed in the HTML application. Continuing the example above, the identified structures can be the customer name, home address and other input fields of the auction page.

Related aspects of the invention provide methods as discussed above, in which the parsing step includes searching for patterns of similarly formatted HTML elements, such as classnames, font size, style, tag color, and size. The parsing step then generates a list of UI objects that represents the displayed items.

In another aspect, the invention provides methods as discussed above that include determining the URLs (or portions thereof) underlying the HTML pages displayed by the browser. Those URLs (or portions) are used to facilitate identifying and distinguishing among UI objects contained on those pages.

Further aspects of the invention provide methods for generating programming instructions or other code embodying the foregoing. Such methods can include creating a source input file containing the aforementioned grammar, e.g., which can serve as an input to an automated parser generator tool.

Yet still further aspects of the invention provide methods as described above for use with markup languages other than HTML, such as XHTML and XUL, by way of non-limiting example.

Further aspects of the invention provide digital data processing systems operating in accord with the above-described method.

Still further aspects of the invention provide systems paralleling the operation described above. These and other aspects of the invention are evident in the drawings, description, and claims that follow.

Brief Description of the Drawings

A more complete understanding of the invention may be attained by reference to the drawings, in which:

Figure 1 depicts the architecture of a digital data processing system that utilizes a grammar-based user interface object recognition system according to the invention;

Figure 2 is a flow chart that depicts generating a computer program according to the invention;

Figure 3 is a flow chart that depicts the operation of the generated program of Figure 2;

Figure 4 depicts the generation and operation of a program according to the invention using YACC to generate C++ code for the generated program;

Figure 5 is a graphical representation of the generated program as a stand-alone module that accepts HTML DOM and outputs user interface objects;

Figure 6 is a graphical representation of further workings of the generated computer program, including the scanner module, parser module and object creation;

Figure 7 depicts a screenshot of a hypothetical HTML application.

Detailed Description

The present invention provides methods and systems for recognizing user interface objects in HTML applications, as well as for creating computer programs that comprise those methods and systems. Computer programs built using the disclosed method accept HTML Document Object Model (DOM) structures as input and provide a list of user interface (UI) objects as output. Alternative methods utilize HTML streams in lieu of, or in addition to, the HTML DOM. Regardless, the approach to UI object recognition is grammar-based.

At the outset, it will be appreciated that DOM is a platform-neutral and language-neutral interface that allows computer programs and scripts to dynamically access and update the content, structure and style of web documents. DOM is a World Wide Web Consortium (W3C) recommended specification. Internet Explorer, Netscape Navigator and all other DOM-compliant browsers implement the interfaces and functions defined by the DOM specification so that programs and scripts can access and update the HTML content loaded into the browser. When the browser loads an HTML document, it creates an internal runtime representation of it. Methods and systems according to the invention capitalize on the DOM interfaces exposed by the browsers, to access this runtime representation of the HTML document.

The content and structure of the DOM is similar to the original HTML. Thus, the DOM is a hierarchical structure of the HTML tags (called HTML elements in DOM terminology) listing all the HTML attributes. In this light, though the illustrated embodiment processes the current state of the HTML document by accessing the document's runtime representation, alternative embodiments may process the HTML stream directly. The context of a displayed page, e.g., as indicated by URL, is also utilized to facilitate applying grammar interpretation. Applications that do not provide this distinguishing characteristic require empirical examination to identify other distinguishing features that determine context.

Those skilled in the art will understand that accessing the runtime state of the document through DOM, and not its HTML source, has certain advantages. First, certain properties of documents are only available when they are displayed by a specific browser. Examples of such

properties include, the exact and current screen position of rendered elements, the current state of a check box, the current content of an edit field, etc. Second, documents are also dynamic structurally, that is, they can be changed by script interfaces such as JavaScript and VBScript that are embedded in the HTML text. This allows new HTML tags to be inserted, existing tags to be modified or removed as the scripts respond to user interactions.

In view of the foregoing, FIGURE 1 depicts an environment 10 in which the invention can be utilized. A digital data processor 16 "serves" HTML applications over a network 14. This can be an intranet, extranet, Internet or other network that supports HTML applications. Web browsers executing on client digital data processors 12 and 18 process HTML streams received from the server 16 to display text, pictures, video, hyperlinks and other output, while accepting text, user selections and other input, all in the conventional manner. In the illustrated embodiment, scanner module 22, parser 24 and other functionality executing on or in connection with the client devices 12, 18 (or, optionally, in connection with the server 16) access the HTML DOM structure to identify UI objects displayed by the browsers executing on those respective devices.

To this end, illustrated scanner module 22 traverses the DOM and creates one or more tokens for each element encountered. The process involves grouping sequences of attributes from the HTML DOM into units (tokens) to create a mapping between the DOM elements and the tokens. The generated tokens, which represent the DOM elements, are then passed to the parser 24.

The tokenized input is interpreted by the parser 24 according to an application specific grammar, to identify and distinguish among the various portions of the application's display. For example, recognizing interactive objects as distinct from textual titles, metatags, unprintable HTML, in-line pictures, etc. The parser can achieve this by grouping the tokens into syntactic structures that identify items displayed in the HTML application. The parser 24 then outputs user interface objects 26 that correspond to graphical elements and other items displayed by the browser.

FIGURE 2 is a flow chart 28 that depicts various steps for implementing this exemplary embodiment of the method of the invention. In step 30, the DOM of a specific application is analyzed and an application-specific grammar is defined. It is important to note that the defined grammar is not a generic grammar for all HTML DOM applications, rather, the defined grammar is specific to a particular application only.

Web designers usually reveal something about their web pages by using style-sheets and attaching style-classes to HTML tags. Such items are useful in defining the grammar of a specific application. Style-names are usually based on the purpose of the HTML tags, because the reason for using style-sheets is to ensure a common look-and-feel of web pages.

For example, a generic SPAN tag, which is used to create a button-like user interface element on the browser, might be given a certain classname by specifying the class attribute for the SPAN tag to ensure that all buttons look the same. This gives a very useful hint about the use of the specific HTML tag. If something like <SPAN CLASS=button> is identified during traversal of the DOM hierarchy, then this construct would be identified as a button-like user interface element on the browser.

The grammar can be defined for a particular application based on this and similar information. If the particular application does not utilize style-sheets, the DOM hierarchy is parsed searching for patterns of similarly formatted HTML elements such as font size, style, tag color, size, etc.

There is a close correlation between the look and intended use of HTML tags. The task is to distinguish between the various user interface elements on the browser by filtering, joining, and classifying HTML tags into user interface objects. Web designers do the same task in the opposite direction. They have certain user interface objects that they turn into HTML tags. They might choose to use cascading style-sheets for this purpose or they might format HTML tags individually.

The end result is the same, user interface objects of a certain class will produce a recurring pattern of HTML tags in the document. The method of the invention can be used to build a parser that recognizes these patterns based on style-sheet classes or patterns of similarly formatted HTML elements.

In step 32, a source file is created for the well known parser generator, Yet Another Compiler-Compiler (YACC), though practice of the invention is not so limited. Those skilled in the art are fully aware that parsers can be hand-programmed or built with other commonly available tools. YACC produces a parser which is a C/C++ program. This C/C++ program can be compiled and linked with other source code modules, including scanner and DOM traversal modules.

The YACC input source file contains an application specific grammar together with C++ statements for outputting UI objects. Those of ordinary skill in the art are very familiar with the workings and notation of YACC, however the following peripheral coverage may be useful. The definitions section of the source file can contain the typical things found at the top of a C program: constant definitions, structure declarations, include statements, and user variable definitions. Tokens may also be defined in this section.

The rules section of the YACC source file contains the application-specific grammar. The left-hand side is separated from the right-hand side by a colon. The actions may appear interspersed in the right-hand side; this means they will be executed when all the tokens or nonterminals to the action's left are shifted. If the action occurs after a production, it will be invoked when a reduction of the right-hand side to the left-hand side of a production is made.

The following is a non-limiting example of a parser built for a hypothetical HTML application. Figure 7 depicts a screenshot of the application. Presented below are the underlying HTML source, and an annotated YACC source file excerpt with C++ action code.

HTML source

The following HTML source does not contain scripting code or form support as those elements of the HTML application are irrelevant for the purposes of demonstration.

```
<html>
 <head>
  <title>
   User Interface
  </title>
  <style type="text/css">
   . { font-size: 10pt; font-family: arial,sans-serif }
   .uibody { }
   .uimenubar { background-color: #acacac }
   .uimenuitem { font-size: 9pt; padding-right: 6px; padding-
left: 6px }
   .uitoolbar { background-color: #c0c0c0 }
   .uipanel { }
   .uibtn { width: 16px; border-top: 1px solid; border-bottom:
1px solid; border-left: 1px solid; border-right: 1px solid }
   .uibtntxt { nowrap; border-top: 1px solid; border-bottom: 1px
solid; border-left: 1px solid; border-right: 1px solid }
   .uigroup { background-color: #e0e0e0; border-top: 1px solid;
border-bottom: 1px solid; border-left: 1px solid; border-right:
1px solid }
  </style>
 </head>
 <body class="uibody" leftmargin="0" topmargin="0">
  <table cols="1" cellspacing="0" cellpadding="0" width="100%"
height="100%">
   <tr>
    <td>
     <table cols="2" cellpadding="0" cellspacing="0"
width="100%">
      <tr>
       <td class="uimenubar" height="20">
        <table>
         <tr>
          <td class="uimenuitem" nowrap>
           File
          </td>
          <td class="uimenuitem" nowrap>
           Edit
          </td>
          <td class="uimenuitem" nowrap>
           View
          </td>
          <td class="uimenuitem" nowrap>
```

```
                Insert
               </td>
               <td class="uimenuitem" nowrap>
                Tools
               </td>
               <td class="uimenuitem" nowrap>
                Topic
               </td>
               <td class="uimenuitem" nowrap>
                Help
               </td>
               <td width="100%" align="right">
               </td>
              </tr>
             </table>
            </td>
           </tr>
           <tr>
            <td height="1" bgcolor="#000000">
            </td>
           </tr>
           <tr>
            <td class="uitoolbar" height="28">
             <table>
              <tr>
               <td class="uibtn" align="middle">
                <span id="b01"><img src="01.gif" border="0"
alt="Create Title"></span>
               </td>
               <td class="uibtn" align="middle">
                <span id="b02"><img src="02.gif" border="0"
alt="Open Title"></span>
               </td>
               <td class="uibtn" align="middle">
                <span id="b03"><img src="03.gif" border="0"
alt="Publish Content"></span>
               </td>
               <td width="4">
               </td>
               <td class="uibtn" align="middle">
                <span id="b04"><img src="04.gif" border="0"
alt="Cut"></span>
               </td>
               <td class="uibtn" align="middle">
                <span id="b05"><img src="05.gif" border="0"
alt="Copy"></span>
               </td>
```

```
            <td class="uibtn" align="middle">
             <span id="b06"><img src="06.gif" border="0"
alt="Paste"></span>
            </td>
            <td class="uibtn" align="middle">
             <span id="b07"><img src="07.gif" border="0"
alt="Delete"></span>
            </td>
            <td width="4">
            </td>
            <td class="uibtntxt" align="middle">
             <span id="b08"><img src="08.gif"
border="0"> Tool 1</span>
            </td>
            <td class="uibtntxt" align="middle">
             <span id="b09"><img src="09.gif"
border="0"> Tool 2</span>
            </td>
            <td class="uibtntxt" align="middle">
             <span id="b10"><img src="10.gif"
border="0"> Tool 3</span>
            </td>
           </tr>
          </table>
         </td>
        </tr>
        <tr>
         <td height="1" bgcolor="#000000">
         </td>
        </tr>
       </table>
      </td>
     </tr>
     <tr>
      <td class="uipanel" width="100%" height="100%">
       <div style="position: relative; left: 0; top: 0; width:
100%; height: 100%; overflow: auto">
        <table cellpadding="5" cellspacing="5">
         <tr>
          <td class="uigroup">
           <div style="position: relative; left: 0; top: 0;
width: 100%; height: 100%">
            <div class="uicontrol">
             <span class="uilabel">Check Boxes</span>
            </div>
            <div class="uicontrol">
             <input type="checkbox" checked>
```

```
                <span class="uilabel">Check 1</span>
               </div>
               <div class="uicontrol">
                <input type="checkbox">
                <span class="uilabel">Check 2</span>
               </div>
               <div class="uicontrol">
                <input type="checkbox">
                <span class="uilabel">Check 3</span>
               </div>
              </div>
             </td>
             <td class="uigroup">
              <div style="position: relative; left: 0; top: 0;
width: 100%; height: 100%">
               <div class="uicontrol">
                <span class="uilabel">Radio Buttons</span>
               </div>
               <div class="uicontrol">
                <input type="radio" checked>
                <span class="uilabel">Radio 1</span>
               </div>
               <div class="uicontrol">
                <input type="radio">
                <span class="uilabel">Radio 2</span>
               </div>
              </div>
             </td>
             <td class="uigroup">
              <div style="position: relative; left: 0; top: 0;
width: 100%; height: 100%">
               <div class="uicontrol">
                <span class="uilabel">Buttons</span>
               </div>
               <div class="uicontrol">
                <input type="button" value="Push 1">
               </div>
               <div class="uicontrol">
                <input type="button" value="Push 2">
               </div>
              </div>
             </td>
             <td class="uigroup">
              <div style="position: relative; left: 0; top: 0;
width: 100%; height: 100%">
               <p>
```

```
                This is sample text with <a
href="http://www.yahoo.com/">Link 1</a>, <a
href="http://www.msn.com/">Link 2</a> and <a
href="http://www.aol.com/">Link 3</a>.
              </p>
             </div>
            </td>
           </tr>
          </table>
          <table cellpadding="5" cellspacing="5">
           <tr>
            <td class="uigroup">
             <div style="position: relative; left: 0; top: 0;
width: 100%; height: 100%">
              <div class="uicontrol">
               <span class="uilabel">Edits, Combo Boxes, List
Boxes</span>
              </div>
              <table>
               <tr>
                <td>
                 <span class="uilabel">Text 1</span>
                </td>
                <td>
                 <input type="text">
                </td>
               </tr>
               <tr>
                <td>
                </td>
                <td>
                 <input type="text">
                </td>
               </tr>
               <tr>
                <td>
                 <span class="uilabel">Combo 1</span>
                </td>
                <td>
                 <select class="uicombo" size="1">
                  <option value="1">
                   Option 1
                  </option>
                  <option value="2">
                   Option 2
                  </option>
                  <option value="3">
```

```
                    Option 3
                   </option>
                  </select>
                 </td>
                </tr>
                <tr>
                 <td>
                  <span class="uilabel">Text 2</span>
                 </td>
                 <td>
                  <input type="text">
                 </td>
                </tr>
               </table>
              </div>
             </td>
             <td class="uigroup">
              <div style="position: relative;left: 0;top: 0;width:
100%;height: 100%">
               <table>
                <tr>
                 <td>
                  <span class="uilabel">Combo 2</span>
                 </td>
                 <td>
                  <select class="uicombo" size="1">
                   <option value="1">
                    Option 1
                   </option>
                   <option value="2">
                    Option 2
                   </option>
                   <option value="3">
                    Option 3
                   </option>
                  </select>
                 </td>
                </tr>
                <tr>
                 <td>
                  <span class="uilabel">Text 3</span>
                 </td>
                 <td>
                  <input type="text">
                 </td>
                </tr>
                <tr>
```

```
                    <td>
                     <span class="uilabel">List 1</span>
                    </td>
                    <td>
                     <select class="uilistbox" size="3">
                      <option value="1">
                       Option 1
                      </option>
                      <option value="2">
                       Option 2
                      </option>
                      <option value="3">
                       Option 3
                      </option>
                     </select>
                    </td>
                   </tr>
                  </table>
                 </div>
                </td>
               </tr>
              </table>
             </div>
            </td>
           </tr>
          </table>
         </body>
        </html>
```

YACC example

```
%token A
%token BODY_UIBODY
%token IMG
%token INPUT_button
%token INPUT_checkbox
%token INPUT_radio
%token INPUT_text
%token SELECT_UICOMBO
%token SELECT_UILISTBOX
%token SPAN
%token SPAN_UILABEL
%token TD
%token TD_UIBTN
%token TD_UIBTNTXT
%token TD_UIMENUBAR
%token TD_UIMENUITEM
```

```
%token TD_UITOOLBAR
%token TD_UIPANEL

%%
```
body

The symbol "body" is the starting symbol of the grammar and it refers to the BODY HTML tag. In fact the right hand side of the rule begins with the recognition of a BODY HTML tag, as the first DOM element that is analyzed is always this tag.

Notice the use of error tokens throughout the rules denoting areas of the input that are ignored. The illustrated system takes advantage of the fact that the yacc generated LALR(1) parser uses an error handling and recovery mechanism that is suitable for ignoring certain parts of the input stream.

```
        : BODY_UIBODY error TD_UIMENUBAR menu TD_UITOOLBAR
toolbar TD_UIPANEL panel
```

The symbol stack of the parser either contains references to DOM elements in case of terminal symbols (tokens representing HTML tags, such as TD_UIMENUBAR above) or a pointer to the tree of generated UI objects in case of non-terminal symbols (such as menu, toolbar and panel above). The step below concatenates two trees of UI objects and puts the result in m_sResult, the output of the parser. Notice the fact that this action gets executed as a final step of the syntactical analysis, when this rule is reduced to the starting symbol.

Here, the tokens based on TD tags are to partition the input stream. The first TD tag with the 'TD_UIMENUBAR' classname attribute signals the beginning of the menu bar (menu). This is one example, where this grammar-based approach shows its strength: it is very easy to create rules that are applied locally (to a certain part of the input stream) and the context is always well defined and followed by the parser.

```
{
   CParserSymbol *t;
   m_sResult = JoinAsSiblings(t = JoinAsSiblings($4, $6), $8);
   m_iResult = 1;
   $$ = NULL;
}
| error
{
   m_sResult = NULL;
   m_iResult = 0;
   $$ = NULL;
```

```
}
;
```

The following two groups of rules (menu, menu_item) recognize the menu (as shown on the top of the browser client area on the screenshot). The first rule group makes processing of several menu items possible. It is one way to express iteration of items in the grammar.

```
menu
   : menu_item
   {
      $$ = $1;
   }
      | menu menu_item
```

The created UI objects must be joined in a tree, that is the purpose of the JoinAsSiblings function.

```
   {
      $$ = JoinAsSiblings($1, $2);
   }
      ;

menu_item
   : TD_UIMENUITEM
```

The constructor of the class CGenericObject is the place where the list of UI object identifiers that correspond to the graphical elements displayed in the browser is generated. (See Figure 3, item 48). Inside the constructor the parser is able to access various properties of a DOM element, in this case its innerText and clientRects properties. In this simple instance there is a one-to-one relationship between the DOM element (the TD tag that is the only symbol on the right side of the rule – this element of the symbol stack is accessed with the $1) and the UI object.

```
   {
      $$ = new CGenericObject(this, NULL, NULL, L"MENUITEM",
L"ROLE_SYSTEM_MENUITEM", 0, 0, 0, 0, -1, 1, $1);
   }
   | error
```

Anything other than the above TD_UIMENUITEM tags are ignored by this error alternative.

```
   {
      $$ = NULL;
   }
   ;
```

Iteration over toolbar buttons is done the same way as for menu items.

```
toolbar
   : toolbar_item
   {
      $$ = $1;
   }
   | toolbar toolbar_item
   {
      $$ = JoinAsSiblings($1, $2);
   }
   ;
```

toolbar_item

The same logical UI abstraction (a button) might be implemented in many different ways even in one application. These alternatives must be present here in the grammar. These rules also present examples of logical UI objects that are created from more than one DOM element. The clientRects attribute is read from the SPAN tag while the logical name of the object might be found as the ALT attribute of the IMG tag (first alternative) or as innerText of the SPAN tag (second alternative).

```
   : TD_UIBTN '{' SPAN '{' IMG '}' '}'
   {
      $$ = new CGenericObject(this, NULL, NULL, L"PUSHBUTTON",
L"ROLE_SYSTEM_PUSHBUTTON", -1, 0, 0, 0, -1, 2, $1, $5);
      CComVariant vAlt;
      CComBSTR sAlt("alt");
      ((CPSHTMLElement*)$5)->m_pElement->getAttribute(sAlt, 0,
&vAlt);
      if (vAlt.vt == VT_BSTR)
         ((CBXObjectTree*)$$)->m_OD.odName =
::SysAllocString(vAlt.bstrVal);
      }
   | TD_UIBTNTXT '{' SPAN '{' IMG '}' '}'
```

This alternative only differs from the 1st in the classname of the SPAN element. These classnames are mapped to different tokens although they can be unified in the scanner/tokenizer. This makes the parser more complex but makes the creation of UI objects simpler.

```
   {
         $$ = new CGenericObject(this, NULL, NULL, L"PUSHBUTTON",
L"ROLE_SYSTEM_PUSHBUTTON", 0, 0, 0, 0, -1, 2, $1, $5);
   }
   | error
   {
      $$ = NULL;
   }
```

```
;
```

Iteration over the rest of the controls is done the same way as for menu items.

```
panel
   : panel_item
   {
      $$ = $1;
   }
   | panel panel_item
   {
      $$ = JoinAsSiblings($1, $2);
   }
   ;
```

The following groups of rules describe the various controls that can be found in the example application.

```
panel_item
   : A
   {
      $$ = new CGenericObject(this, NULL, NULL, L"LINK",
L"ROLE_SYSTEM_LINK", 0, 0, 0, 0, -1, 1, $1);
   }
   | INPUT_button
   {
      $$ = new CGenericObject(this, NULL, NULL, L"PUSHBUTTON",
L"ROLE_SYSTEM_PUSHBUTTON", -1, 0, 0, 0, -1, 1, $1);
      CComVariant vValue;
      CComBSTR sValue("value");
      ((CPSHTMLElement*)$1)->m_pElement->getAttribute(sValue, 0,
&vValue);
      if (vValue.vt == VT_BSTR)
         ((CBXObjectTree*)$$)->m_OD.odName =
::SysAllocString(vValue.bstrVal);
   }
   | INPUT_checkbox SPAN_UILABEL
   {
      $$ = new CGenericObject(this, NULL, NULL, L"CHECKBUTTON",
L"ROLE_SYSTEM_CHECKBUTTON", 1, 0, 0, 0, -1, 2, $1, $2);
   }
   | INPUT_radio SPAN_UILABEL
   {
      $$ = new CGenericObject(this, NULL, NULL, L"RADIOBUTTON",
L"ROLE_SYSTEM_RADIOBUTTON", 1, 0, 0, 0, -1, 2, $1, $2);
   }
   | panel_item_nl
```

```
   {
      $$ = $1;
   }
   | TD '{' SPAN_UILABEL '{' '}' '}' TD '{' panel_item_nl
```

This rule assigns labels to controls. panel_item_nl is a non-terminal which means that the symbol stack contains an already recognized control's object at the time of the execution of the action. The action itself copies the innerText of the second SPAN tag to the name member of the control's UI object.

```
   {
      ((CPSHTMLElement*)$3)->m_pElement-
>get_innerText(&((CBXObjectTree*)$9)->m_OD.odName);
      $$ = $9;
   }
   | error
```

This alternative makes the parser to ignore any unimportant input. Any sequence of tokens that are different from the previous alternatives are skipped.

```
   {
      $$ = NULL;
   }
   ;

panel_item_nl
   : INPUT_text
   {
      $$ = new CGenericObject(this, NULL, NULL, L"TEXT",
L"ROLE_SYSTEM_TEXT", -1, 0, 0, 0, -1, 1, $1);
   }
   | SELECT_UICOMBO
   {
      $$ = new CGenericObject(this, NULL, NULL, L"COMBOBOX",
L"ROLE_SYSTEM_COMBOBOX", -1, 0, 0, 0, -1, 1, $1);
   }
   | SELECT_UILISTBOX
   {
      $$ = new CGenericObject(this, NULL, NULL, L"LIST",
L"ROLE_SYSTEM_LIST", -1, 0, 0, 0, -1, 1, $1);
   }
   ;

%%
```

C++ helper classes

```
typedef struct tagSBXObjectDescriptor
{
   BSTR odClass;
   BSTR odLogicalClass;
   BSTR odName;
   BSTR odLabel;
   DWORD   odStyle;
   DWORD   odStatus;
   DWORD   odFlags;
   ULONG   uRectCount;
   [size_is(uRectCount)]
   RECT *odRect;
   BSTR odContents;
} SBXObjectDescriptor;

class CParserSymbol
{
public:
   CParserSymbol();
   virtual ~CParserSymbol();
};

class CPSHTMLElement : public CParserSymbol
{
public:
   CPSHTMLElement(const CHTMLElementPtr&, int iToken);
   ~CPSHTMLElement();
   CHTMLElementPtr m_pElement;
   int m_iToken;
};

class CBXObjectDescriptor : public SBXObjectDescriptor
{
public:
   CBXObjectDescriptor();
   ~CBXObjectDescriptor();
   SBXObjectDescriptor *pSBXObjectDescriptor);
};

class CBXObjectTree : public CParserSymbol
{
public:
   CBXObjectTree(CBXObjectTree*, CBXObjectTree*);
   ~CBXObjectTree();
   CBXObjectTree *m_pChild;
   CBXObjectTree *m_pSibling;
   CBXObjectDescriptor m_OD;
```

```
};

class CGenericObject : public CBXObjectTree
{
public:
   CGenericObject(yyfparser*, CBXObjectTree*, CBXObjectTree*);
   CGenericObject(yyfparser*, CBXObjectTree*, CBXObjectTree*,
LPCWSTR szClass, LPCWSTR szLogClass, LONG uName, DWORD dwStatus,
LONG uStartRect, LONG uEndRect, LONG uContents, ULONG
uParamCount, ...);
   ~CGenericObject();
   void FillRectArray(const CHTMLElement2Ptr &pElement2, const
POINT &pt);

   CElementArray m_aParams;
   CDWordArray m_aParamTokens;
   LONG m_uStartRect;
   LONG m_uEndRect;
   LONG m_uContents;
   CDWord2PtrMap m_mElementIndex;

   void Construct(LPCWSTR szClass, LPCWSTR szLogClass, LONG
uName, DWORD dwStatus, LONG uStartRect, LONG uEndRect, LONG
uContents, ULONG uParamCount, va_list) ;
};

CParserSymbol *JoinAsSiblings(CParserSymbol *&p1, CParserSymbol
*&p2)
{
   CBXObjectTree *pTree1 = (CBXObjectTree*)p1;
   if (p1 && p2) {
      while (pTree1->m_pSibling) {
         pTree1 = pTree1->m_pSibling;
      }
      pTree1->m_pSibling = (CBXObjectTree*)p2;
   }
   CParserSymbol *r = p1 ? p1 : p2;
   p1 = NULL;
   p2 = NULL;
   return r;
}

CParserSymbol *JoinAsDescendants(CParserSymbol *&p1,
CParserSymbol *&p2)
{
   CBXObjectTree *pTree1 = (CBXObjectTree*)p1;
   if (p1 && p2) {
```

```
      while (pTree1->m_pChild) {
         pTree1 = pTree1->m_pChild;
      }
      pTree1->m_pChild = (CBXObjectTree*)p2;
   }
   CParserSymbol *r = p1 ? p1 : p2;
   p1 = NULL;
   p2 = NULL;
   return r;
}

CParserSymbol::CParserSymbol()
{
}

CParserSymbol::~CParserSymbol()
{
}

CPSHTMLElement::CPSHTMLElement(const CHTMLElementPtr &pElement,
int iToken)
: m_pElement(pElement), m_iToken(iToken)
{
}

CPSHTMLElement::~CPSHTMLElement()
{
}

CBXObjectDescriptor::CBXObjectDescriptor()
{
   ::ZeroMemory(this, sizeof(SBXObjectDescriptor));
}

CBXObjectDescriptor::~CBXObjectDescriptor()
{
   if (odClass)
      ::SysFreeString(odClass);
   if (odLogicalClass)
      ::SysFreeString(odLogicalClass);
   if (odName)
      ::SysFreeString(odName);
   if (odLabel)
      ::SysFreeString(odLabel);
   if (odContents)
      ::SysFreeString(odContents);
   if (odRect)
```

```
    delete odRect;
}

CBXObjectTree::CBXObjectTree(CBXObjectTree *pChild,
CBXObjectTree *pSibling)
: m_pChild(pChild), m_pSibling(pSibling)
{
}

CBXObjectTree::~CBXObjectTree()
{
   if (m_pChild)
      delete m_pChild;
   if (m_pSibling)
      delete m_pSibling;
}

CGenericObject::CGenericObject(yyfparser *pParser, CBXObjectTree
*pChild, CBXObjectTree *pSibling)
: CBXObjectTree(pChild, pSibling), m_uStartRect(0), m_uEndRect(-
1), m_uContents(-1)
{

}

CGenericObject::CGenericObject(yyfparser *pParser, CBXObjectTree
*pChild, CBXObjectTree *pSibling, LPCWSTR szClass, LPCWSTR
szLogClass, LONG uName, DWORD dwStatus, LONG uStartRect, LONG
uEndRect, LONG uContents, ULONG uParamCount, ...)
: CBXObjectTree(pChild, pSibling), m_uStartRect(0), m_uEndRect(-
1), m_uContents(-1)
{
   va_list argList;
   va_start(argList, uParamCount);
   Construct(szClass, szLogClass, uName, dwStatus, uStartRect,
uEndRect, uContents, uParamCount, argList);
   va_end(argList);
}

CGenericObject::~CGenericObject()
{
}

void CGenericObject::Construct(LPCWSTR szClass, LPCWSTR
szLogClass, LONG uName, DWORD dwStatus, LONG uStartRect, LONG
uEndRect, LONG uContents, ULONG uParamCount, va_list argList)
{
```

```
   m_uStartRect = uStartRect;
   m_uEndRect = uEndRect;
   m_uContents = uContents;

   CParserSymbol *pElement;
   if (uParamCount)
   {
      for (ULONG i = 0; i < uParamCount; i++)
      {
         pElement = va_arg(argList, CParserSymbol*);
         m_aParams.push_back(((CPSHTMLElement*)pElement)-
>m_pElement);
         m_aParamTokens.push_back(((CPSHTMLElement*)pElement)-
>m_iToken);
         CUnknownPtr pElementUnk(((CPSHTMLElement*)pElement)-
>m_pElement);

   m_mElementIndex.insert(CDWord2PtrMap::value_type((DWORD)(IUnkn
own*)pElementUnk, (LPVOID)i));
      }
   }

   m_OD.odClass = ::SysAllocString(szClass);
   m_OD.odLogicalClass = ::SysAllocString(szLogClass);
   m_OD.odStatus = dwStatus;

   if (uParamCount)
   {
      if (uName >= 0)
      {
         m_aParams[uName]->get_innerText(&m_OD.odName);
         CropWhitespace(m_OD.odName);
      }
   }
   if (m_OD.odRect)
      delete m_OD.odRect;
   m_OD.odRect = NULL;
   m_OD.uRectCount = 0;
   for (LONG i = m_uStartRect; i <= m_uEndRect; i++)
      FillRectArray(CHTMLElement2Ptr(m_aParams[i]), ptOffset);
}


void CGenericObject::FillRectArray(const CHTMLElement2Ptr
&pElement2, const POINT &ptOffset)
{
   if (!pElement2) {
```

```
      return ;
   }
   CHTMLRectCollectionPtr pRectCollection;
   pElement2->getClientRects(&pRectCollection);
   if (pRectCollection) {
      ULONG uRectCount = m_OD.uRectCount;
      pRectCollection->get_length((long*)&m_OD.uRectCount);
      m_OD.uRectCount += uRectCount;
      RECT *odRect = m_OD.odRect;
      m_OD.odRect = new RECT[m_OD.uRectCount];
      ::ZeroMemory(m_OD.odRect, sizeof(RECT)*m_OD.uRectCount);
      if (odRect) {
         ::CopyMemory(m_OD.odRect, odRect,
sizeof(RECT)*uRectCount);
         delete odRect;
      }
      for (long i = uRectCount; i < (long)m_OD.uRectCount; i++) {
         CComVariant idx = i - (long)uRectCount, rval;
         pRectCollection->item(&idx, &rval);
         if (rval.vt == VT_DISPATCH) {
            CHTMLRectPtr pRect = rval.pdispVal;
            if (pRect) {
               pRect->get_left(&m_OD.odRect[i].left);
               pRect->get_top(&m_OD.odRect[i].top);
               pRect->get_right(&m_OD.odRect[i].right);
               pRect->get_bottom(&m_OD.odRect[i].bottom);
            }
         }
      }
   }
}
```

Referring again to FIGURE 2, in step 34, YACC is run with its input source file to generate C++ source code. The code generated by YACC can be altered before compiling and executing. The parser produced by YACC is a LALR(1) parser. A LR(1) parser can also be used instead. In these regards, those skilled in the art will appreciate that an LR(1)/LALR(1) grammar is the preferred, because it supports processing of an input stream in one pass. This characteristic results in increased performance (high speed), e.g., of the type necessary for use with a real-time application. YACC is an acceptable tool for use herein because it readily supports LR(1)/LALR(1) grammars.

In step 36, the YACC generated code is combined with the C++ code implementing the other remaining functionality (scanner module, DOM traversal module, etc.) and compiled with a C++ compiler. The resulting program, depicted by step 38, accepts HTML DOM as input and produces a list of UI objects as output that correspond to items and graphical elements displayed in the browser. In an ideal embodiment the generated program is a browser applicable Dynamically Linked Library (DLL).

FIGURE 3 illustrates a flow chart 40 that depicts the various steps of the generated program's operation. In step 42, the program accesses the HTML DOM in a web browser and serializes the tree of DOM elements by traversing the DOM hierarchy. The method of gaining access to the DOM may differ among web browsers. Therefore, the traversal code may need to be implemented somewhat differently depending on the browser being used.

An example that illustrates the above point is the accessing of DOM in the Microsoft Internet Explorer browser. Internet Explorer uses a component object model (COM) based implementation of DOM. By gaining access to the browser's COM interface the invention gains access to the DOM of the currently loaded document. Access to the browser's COM interface is gained by loading a so-called browser helper object into the browser. This is a fine point and is not a limitation of the invention. Those skilled in the art are aware that it is certainly possible to gain access and traverse DOM because of its standardized architecture. The result of the DOM traversal is a sequence of DOM elements with the capability to access the various properties of each element e.g. HTML attributes.

The HTML DOM elements are then processed. In step 44, the program creates one or more tokens for each DOM element. These tokens are used to represent the DOM elements in the parser. The tokens are created by the scanner depicted in Figure 1 as item 22, and in Figure 6 as item 72. Compared to the task of compiling programming languages this is the step of lexical analysis.

A sequence of tokens is created for each DOM element processed, because the aim of the program is to create logical UI objects out of DOM elements. In general, one token is generated

for each DOM element and additional tokens are generated before every first and after every last child of a DOM element. It is possible, although usually not necessary, to create multiple tokens for certain DOM elements. The amount of information that goes into every token is application-dependent. In the case of HTML/CSS-based applications, it is a good starting point to have a unique token per HTML element and the element's classname.

In step 46, the program parses the tokenized input according to the application-specific grammar. Bottom-up left-to-right syntactical analysis is used. Because of the run-time sensitivity of the problem, a LR(1) or preferably LALR(1) parser is built as discussed above.

Finally, as depicted in step 48, the program outputs a list of UI objects that correspond to graphical elements displayed in the browser. It should be noted that the generated program is a stand-alone module that simply inputs HTML DOM and outputs a list of UI objects.

FIGURE 4 is a graphical representation 50 of the overall process. An input file containing an application-specific grammar and source code for outputting UI objects is input into YACC 52. A C++ source code module 54 is then generated. Alternatively, the source code for outputting UI objects can be added to the YACC output file 54 instead of including it in the input file containing the grammar. Additional code modules 56, such as a scanner module or DOM traversal module can then be compiled 58 with the YACC output file 54 to generate the program 60. In a preferred embodiment the executable program 60 is a DLL. As shown in FIGURE 4, the executable program 60 accepts HTML DOM and generates UI objects 62 that correspond to graphical elements in the browser display.

FIGURE 5 is a graphical representation 64 of the executable program 66 as a stand-alone module, which parses a specific HTML DOM and generates UI objects 68. Examples of user interface object properties include: name, content, shape, location, and properties. The invention can be used with software training aids, accessibility aids, and quality assurance for stand-alone, Internet or other computer network-based software applications.

FIGURE 6 depicts a graphical representation 70 of the inner workings of the generated program. The scanner module 72 accesses the HTML DOM and generates tokens for each DOM element that is encountered. These tokens are passed to the parser 74, which then interprets the tokenized input in accordance with the application specific grammar, and generates UI objects 76 that correspond to graphical elements displayed in the browser. In the illustrated embodiment, the application-specific grammar is already pre-defined and "on-board" the parser. However, those skilled in the art will appreciate that the parser can be reconfigured at run-time to reflect at least some changes in the grammar (e.g., using permutations of pre-loaded code snippets).

Illustrated below are C++ code samples that depict the tokens and corresponding HTML elements of a particular application. This code belongs to the scanner 72 part of the system 70. In the list of definitions there is a parser token on the left side associated with a HTML tag name and "classname" on the right side. This mapping is arbitrary and application-specific.

```
const CTokenData HTML_TOKENDATA[] = {
   { A, L"A#" },
   { BODY_UIBODY, L"BODY#uibody" },
   { IMG, L"IMG#" },
   { INPUT_button, L"INPUT_button#"},
   { INPUT_checkbox, L"INPUT_checkbox#"},
   { INPUT_radio, L"INPUT_radio#"},
   { INPUT_text, L"INPUT_text#" },
   { SELECT_UICOMBO, L"SELECT#uicombo" },
   { SELECT_UILISTBOX, L"SELECT#uilistbox" },
   { SPAN, L"SPAN#" },
   { SPAN_UILABEL, L"SPAN#uilabel" },
   { TD, L"TD#" },
   { TD_UIBTN, L"TD#uibtn" },
   { TD_UIBTNTXT, L"TD#uibtntxt" },
   { TD_UIMENUBAR, L"TD#uimenubar" },
   { TD_UIMENUITEM, L"TD#uimenuitem" },
   { TD_UITOOLBAR, L"TD#uitoolbar" },
   { TD_UIPANEL, L"TD#uipanel" }
};
```

The above embodiments are presented for illustrative purposes only. Those skilled in the art will appreciate that various modifications can be made to these embodiments without departing from the scope of the present invention. For example, hand-programmed parsers or

parser generators other than YACC can be utilized and implemented by a system of the invention. By way of further example, the methods and systems according to the invention can be used to process HTML in contexts other than Internet-based, and client-server network applications. Thus, for example, they can be used to interpret UI objects displayed by a browser executing from an HTML file stored directly within a client device, e.g. a stand-alone HTML-based application. By way of further example, the invention can be used with mark-up languages other than HTML, e.g., XHTML, XUL.

In view of the foregoing, what is claimed is: